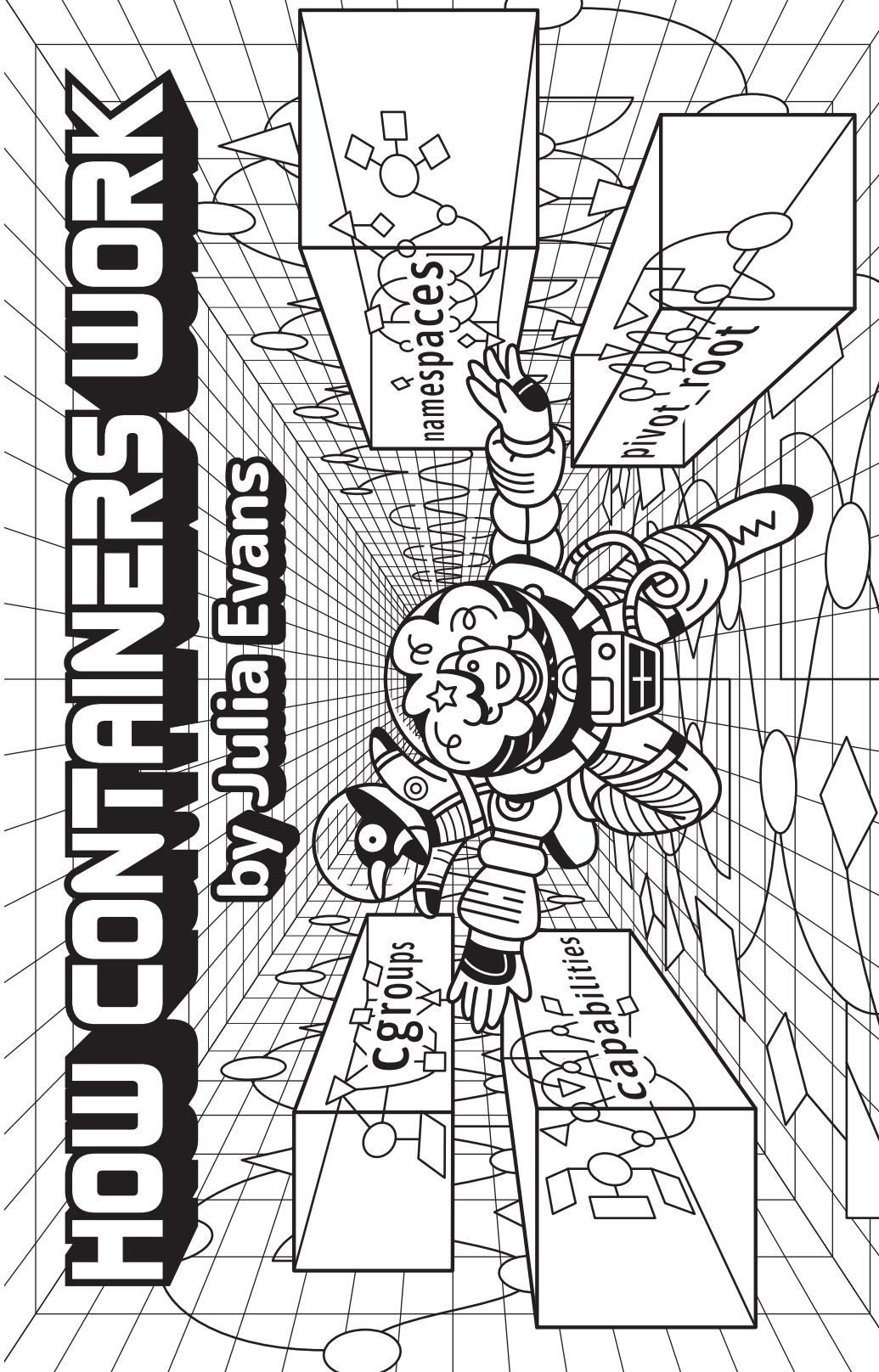


☞ this?
more at
wizardzines.com

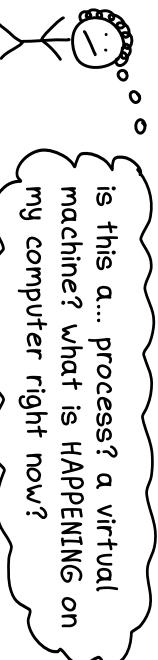
HOW CONTAINERS WORK

by Julia Evans



Why this zine?

When I started using containers I was SO CONFUSED.

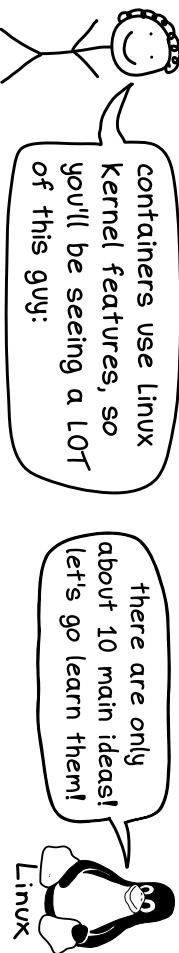


So I decided to learn how they work under the hood!



Now I feel confident that I can solve basically any problem with containers because I understand how they work.

I hope that after reading this zine, you'll feel like that too.



there are only about 10 main ideas!
(let's go learn them!)

♡ thanks for reading ♡

23

I did a bunch of the research for this zine by reading the man pages. But, much more importantly, I experimented -- a lot!

let's see, what happens if I create a cgroup with a memory limit of 5MB?



So, if you have access to a Linux machine, try things out! Mount an overlay filesystem! Create a namespace! See what happens!

[credits](#)

Cover art: Vladimir Kašiković
Editing: Dolly Lanuza, Kamal Marhubi

configuration options

22

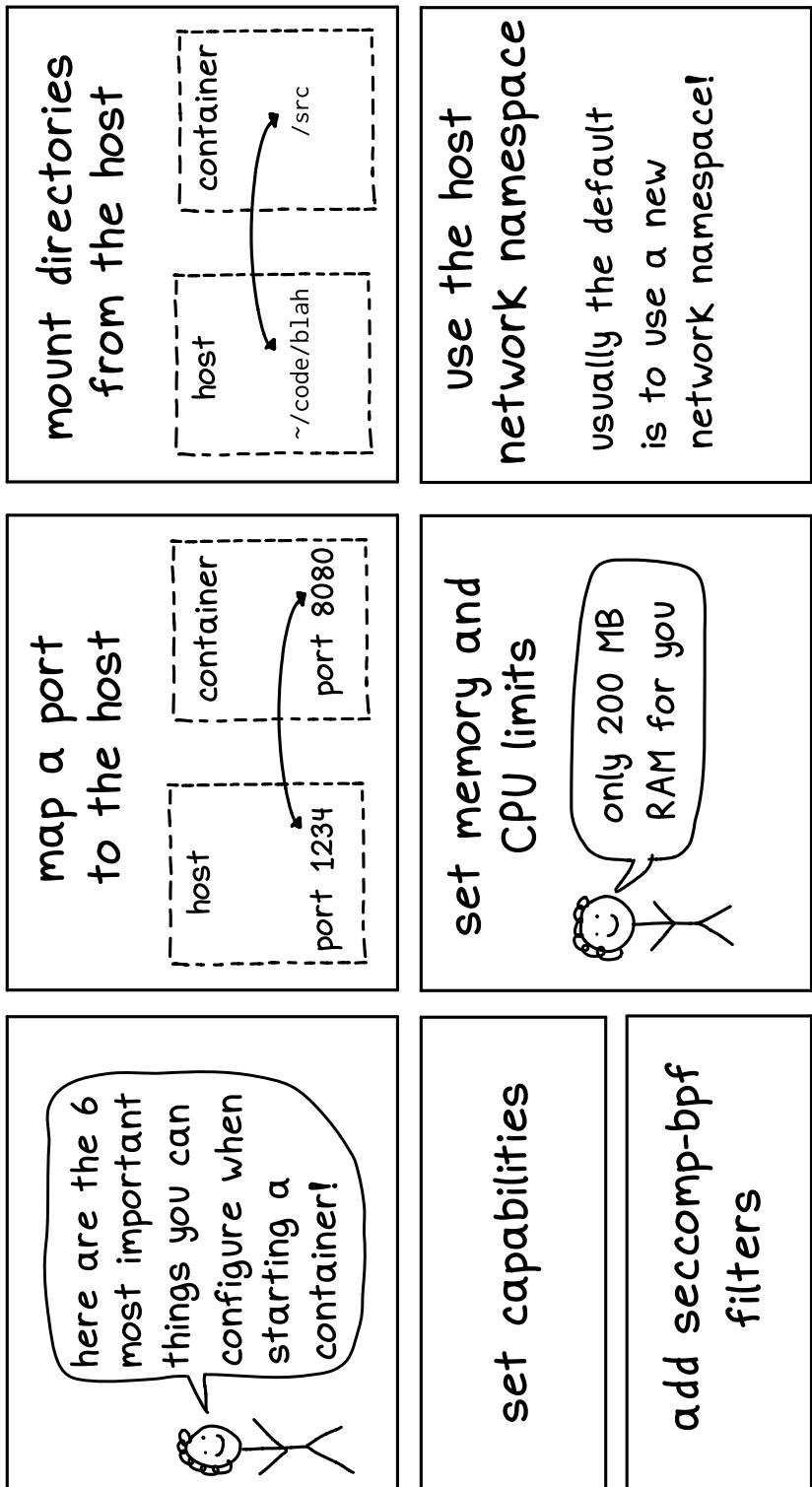


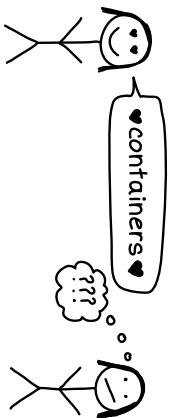
table of contents

why containers?	4	cgroups	13
the big idea: include EVERY dependency	5	namespaces	14
containers aren't magic	6	how to make a namespace	15
containers = processes	7	PID namespaces	16
container kernel features	8	user namespaces	17
pivot_root	9	network namespaces	18
layers	10	container IP addresses	19
overlay filesystems	11	capabilities	20
container registries	12	seccomp-BPF	21
		configuration options	22

Why containers?

4

there's a lot of container hype,



Here are 2 problems they solve...

all containers have their own filesystem

This is the big reason containers are great.

- I'm running Ubuntu 19.04
- I'm running an old CentOS distribution from 2014!
- host OS
- container

```
$ ./configure
$ make all
ERROR: you have version
2.1.1 and you need
at least 2.2.4
```

problem: building software is annoying

solution: package all dependencies in a container ★

Many CI systems use containers.

problem: deploying software is annoying too

solution: deploy a container

seccomp-bpf

seccomp-BPF lets you run a function before every system call

all programs use system calls

rarely-used system calls can help an attacker

Docker blocks dozens of syscalls by default

2 ways to block scary system calls

the function decides if syscall is allowed

example function:

```
if name in allowed_list {
    return true;
} else {
    this means the syscall doesn't happen!
}
```

return false;

↳ this means the syscall doesn't happen!

1. limit the container's capabilities
 2. set a seccomp-bpf whitelist
- You should do both!

capabilities

20

we think of root as being all-powerful...
edit any file
change network config
spy on any program's memory

... but actually to do "root" things, a process needs the right ★capabilities★

I want to modify the route table!
you need CAP_NET_ADMIN

CAP_SYS_ADMIN lets you do a LOT of things. avoid giving this if you can!

CAP_NET_ADMIN allow changing network settings

there are dozens of capabilities
\$ man capabilities
explains all of them but let's go over 2 important ones!

by default containers have limited capabilities
can I call process_vm_readv?
nope! you'd need CAP_SYS_PTRACE for that!

\$ getpcaps PID
print capabilities that PID has
getcap / setcap
system calls: get and set capabilities!

the big idea: include EVERY dependency 5

containers package EVERY dependency together

to make sure this program will run on your laptop, I'm going to send you every single file you need

a container image is a tarball of a filesystem
Here's what's in a typical Rails app's container:

libc + other system libraries
your app's code
Ruby interpreter
Ruby gems
Ubuntu base OS

how images are built
0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM
this is what 'docker build' does!

running an image
1. download the tarball
2. unpack it into a directory
3. run a program and pretend that directory is its whole filesystem
I can set up a Postgres test database in like 5 seconds! wow!

images let you 'install' programs really easily

Containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!
(download this script at bit.ly/containers-arent-magic)

It only runs on Linux because these features are all Linux-only.

```
wget bit.ly/fish-container -O fish.tar
mkdir container-root; cd container-root
tar -xf ./fish.tar
cgroup_id=$(shuf -i 1000-2000 -n 1) # 1. download the image
cgcreate -g "cpu,cpuacct,memory:$cgroup_id" # 2. unpack image into a directory
cgset -r cpu.shares=512 "$cgroup_id" # 3. generate random cgroup name
cgset -r memory.limit_in_bytes=10000000000 # 4. make a cgroup &
                                            # set CPU/memory limits
"$cgroup_id"
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \
unshare -fmuipn --mount-proc \
chroot "$PWD" \
/bin/sh -c "
/bin/mount -t proc proc /proc &&
hostname container-fun-times &&
/usr/bin/fish" # 5. use the cgroup
# 6. make + use some namespaces
# 7. change root directory
# 8. use the right /proc
# 9. change the hostname
# 10. finally, start fish!
```

Container IP addresses

19

Containers often get their own IP address

 I'm running WordPress at 172.17.2.3:8080!
Wordpress container 1

 I'm using 172.17.0.49:8080! Wordpress container 2

Containers use private IP addresses

192.168.*.*	reserved for private networks (RFC 1918)
10.*.*.*	

> 172.16.*.*

This is because they're not directly on the public internet

for a packet to get to the right place, it needs a route

 hi I'm here!

 packet

I don't have any entry matching 172.16.2.3 in my route table, sorry!

inside the same computer, you'll have the right routes

same computer:

```
$ curl 172.16.2.3:8080
```

different computer:

```
$ curl 172.16.2.3:8080
```

... no reply ...

 wow these things change a lot

 route table

cloud providers have systems to make container IPs work

In AWS this is called an "elastic network interface"

network namespaces

18

network namespaces are kinda confusing

- what does it MEAN for a process to have its own network?

namespaces usually have 2 interfaces (+ sometimes more)

- the loopback interface (127.0.0.1/8, for connections inside the namespace)
- another interface (for connections from outside)

every server listens on a port and network interface(s)

- 0.0.0.0:8080 means "port 8080 on every network interface in my namespace"

127.0.0.1 stays inside your namespace

- I'm listening on 127.0.0.1
- That's fine but nobody outside your network namespace will be able to make requests to your server

your physical network card is in the host network namespace

other namespaces are connected to the host namespace with a bridge

7

containers = processes

a container is a group of Linux processes

- on a Mac, all your containers are actually running in a Linux virtual machine

inside the container

```
$ ps aux | grep top
USER PID START COMMAND
root 23540 20:55 top
bork 23546 20:57 top
```

these two are the same process!

container processes can do anything a normal process can ...

- I want my container to do X Y Z W!
- sure! your computer, your rules!

the restrictions are enforced by the Linux Kernel

- NO, you can't have more memory!
- on the next page we'll list all the kernel features that make this work!

Container kernel features

containers use these Linux kernel features

"container" doesn't have a clear definition, but Docker containers use all of these features.

♥ pivot_root ♥

set a process's root directory to a directory with the contents of the the container image

Linux
only 500 MB of RAM for you!

♥ namespaces ♥

allow processes to have their own:

- network → mounts
- PIDs → users
- hostname + more

★ capabilities ★

security: give specific permissions

♥ seccomp-bpf ♥

security: prevent dangerous system calls

★ overlay filesystems ★

this is what makes layers work! Sharing layers saves disk space & helps containers start faster

User namespaces

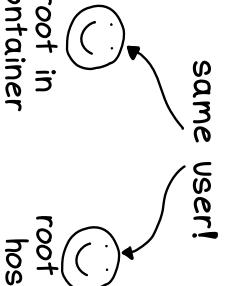
17

user namespaces are a security feature...

I'd like root in the container to be totally unprivileged

you want a user namespace!

... but not all container runtimes use them



"root" doesn't always have admin access

I'm root so I can do ANYTHING right?

container process
(actually you have limited capabilities so mostly you can just access files owned by root!)

in a user namespace, UIDs are mapped to host UIDs

I'm running as UID 0
oh, that's mapped to 12345
Linux process

The mapping is in

/proc/self/uid_map

unmapped users show up as "nobody"

create user namespace
\$ unshare --user bash

\$ ls -l /usr/bin
.. nobody nogroup apropos
.. nobody nogroup apt
.. nobody nogroup apt
but we didn't map any users

how to find out if you have a separate user namespace

compare the results of
\$ ls /proc/PID/ns

between a container process and a host process.

PID namespaces

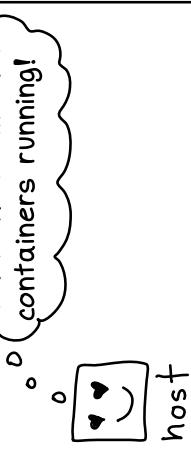
16

the same process has different PIDs in different namespaces

PID in host PID in container
23512 ①
23513 4 PID 1 is
23518 12 special

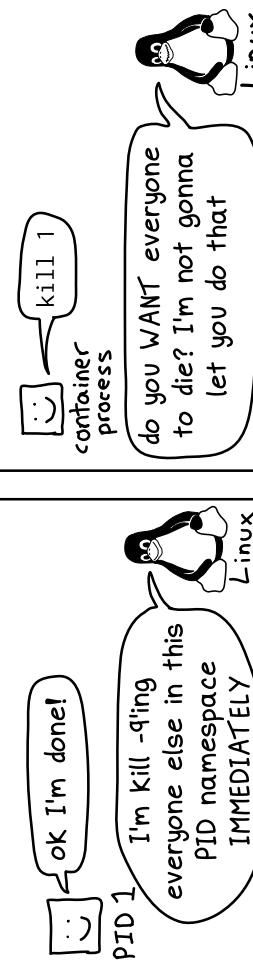
PID namespaces are in a tree

host PID namespace
(the root)

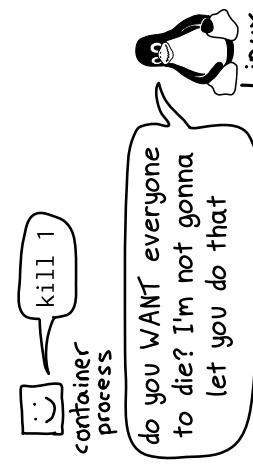


Often the tree is just 1 level deep (every child is a container)

if PID 1 exits, everyone gets killed



Killing PID 1 accidentally would be bad



you can see processes in child PID namespaces

o look at all those containers running!



rules for signaling PID 1

from same container:

only works if the process has set a signal handler

from the host:

only SIGKILL and SIGSTOP are ok, or if there's a signal handler

pivot-root

a container image is a tarball of a filesystem (or several tarballs: 1 per layer)

o if someone sends me a tarball of their filesystem, how do I use that though?

chroot: change a process's root directory

If you chroot to /fake/root, when it opens the file /usr/bin/redis it'll get /fake/root/usr/bin/redis instead.

You can "run" a container just by using chroot, like this:

```
$ mkdir redis; cd redis  
$ tar -xzf redis.tar  
$ chroot $PWD /usr/bin/redis  
# done! redis is running!
```

programs can break out of a chroot

chroot

all these files are still there! A root process can access them if it wants.

redis container directory

redis container directory

to have a "container" you need more than pivot_root

pivot_root alone won't let you:

- set CPU/memory limits
- hide other running processes
- use the same port as another process
- restrict dangerous system calls

Containers use pivot_root instead of chroot.

Layers

different images have similar files

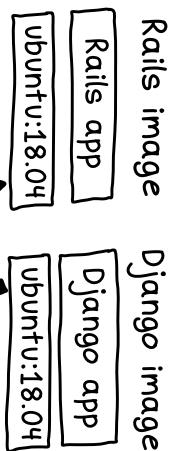


every layer has an ID

usually the ID is a sha256 hash of the layer's contents

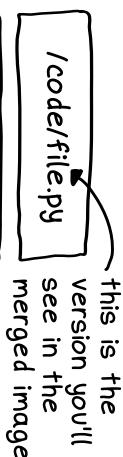
example: 8e99fae2..

reusing layers saves disk space



exact same files on disk!

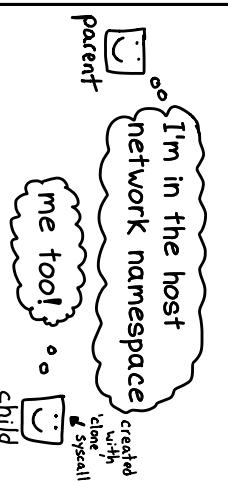
if a file is in 2 layers, you'll see the version from the top layer



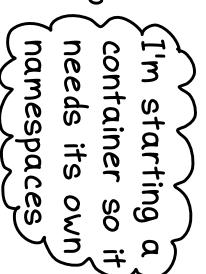
this is the version you'll see in the merged image

how to make a namespace

processes use their parent's namespaces by default



but you can switch namespaces at any time



★clone★ lets you create new namespaces for a child process

- ★clone★ make a new process
- ★unshare★ make + use a namespace
- ★setsns★ use an existing namespace

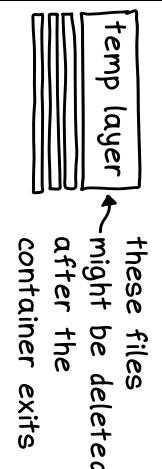
each namespace type has a man page

```
$ man network_namespaces
...
A physical network device can live in exactly one network namespace.
```

a layer is a directory

```
$ ls 8891378eb*
bin/ home/ mnt/ run/ tmp/
boot/ lib/ opt sbin/ usr/
dev/ lib64/ proc/ srv/ var/
etc/ media/ root/ sys/
files in an ubuntu:18.04 layer
```

by default, writes go to a temporary layer



To keep your changes, write to a directory that's mounted from outside the container

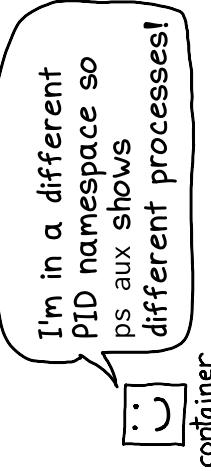
namespaces

14

inside a container,
things look different



why things look different:
↳ namespaces:

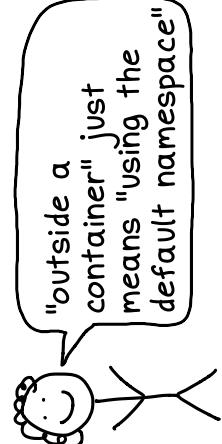


every process has
7 namespaces

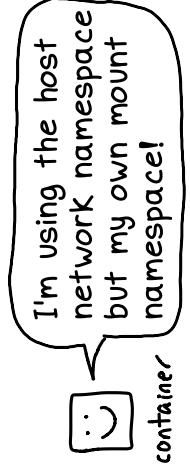
```
$ lsns -p 273
      PID   NS TYPE
4026531835 cgroup
4026531836 pid
4026531837 user
4026531838 uts
4026531839 ipc
4026531840 mnt
4026532009 net
      ↪ namespace ID
```

you can also see a
process's namespace with:
\$ ls -l /proc/273/ns

there's a default
("host" namespace)



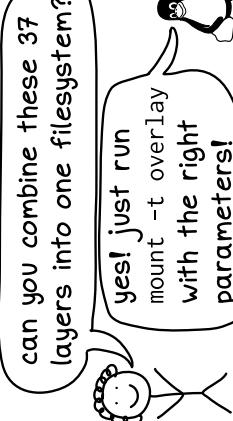
processes can have
any combination
of namespaces



overlay filesystems

11

how layers work:
mount -t overlay



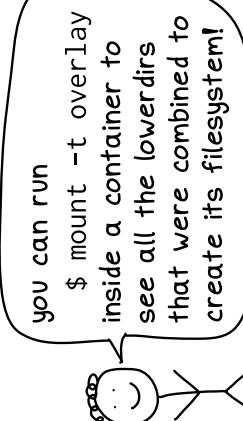
mount -t overlay
has 4 parameters

lowerdir: list of read-only directories
upperdir: directory where writes should go
workdir: empty directory for internal use
target: the merged result

upperdir:
where all writes go

when you create, change, or
delete a file, it's recorded in
the upperdir.
usually this starts out empty
and is deleted when the
container exits

lowerdir:
the layers. read only.



here's an example!

```
$ mount -t overlay overlay -o
  lowerdir=/lower,upperdir=/upper,workdir=/work /merged
$ ls /upper
cat.txt
$ ls /lower
dog.txt
$ ls /merged
bird.txt
      ↪ bird.txt
```

container registries

sharing container images is useful

I made an image you can use to run Redis with just one command!

yay!

there are public container registries...

I'm going to use the latest official public Redis image to test my code!

processes can use a lot of memory

I want 10 GB of memory

me too!

process guys, I only have 16 GB total

Linux

a registry is a server that serves images

images have an ID ^{like} "leffq2" and sometimes a tag

like "18.04" or "latest"

... and private registries

every time we build our web service, we upload a new image to our private registry at COMPANY developer at company

a cgroup is a group of processes

cgroup! every process in a container is in the same cgroup

use too much memory: get OOM killed

"out of memory"

process I want 1 GB of memory

process NOPE your limit was 500 MB you die now!

Linux

use too much CPU: get slowed down

process I want to use ALL THE CPU!

process you hit your quota for this 100ms period, you'll have to wait

Linux

registries let you download just the layers you need

I already have the Ubuntu base image, I just need 0fe223 here's 0fe223! registry

be careful where your container images come from

I'll just run this image from RANDOM_PERSON ... oh no! RANDOM_PERSON is mining bitcoin on my server ... 2 months later ...

cgroups have memory/CPU limits

you three get 500 MB of RAM to share, okay?

cgroup

cgroups track memory & CPU usage

that cgroup is using 112.3 MB of memory right now you can see it in `/sys/fs/cgroup`